# Writing Parsers and Compilers with PLY

David Beazley
http://www.dabeaz.com

February 23, 2007

# Overview

- Crash course on compilers
- An introduction to PLY
- Notable PLY features (why use it?)
- Experience writing a compiler in Python

# Background

- Programs that process other programs
- Compilers
- Interpreters
- Wrapper generators
- Domain-specific languages
- Code-checkers

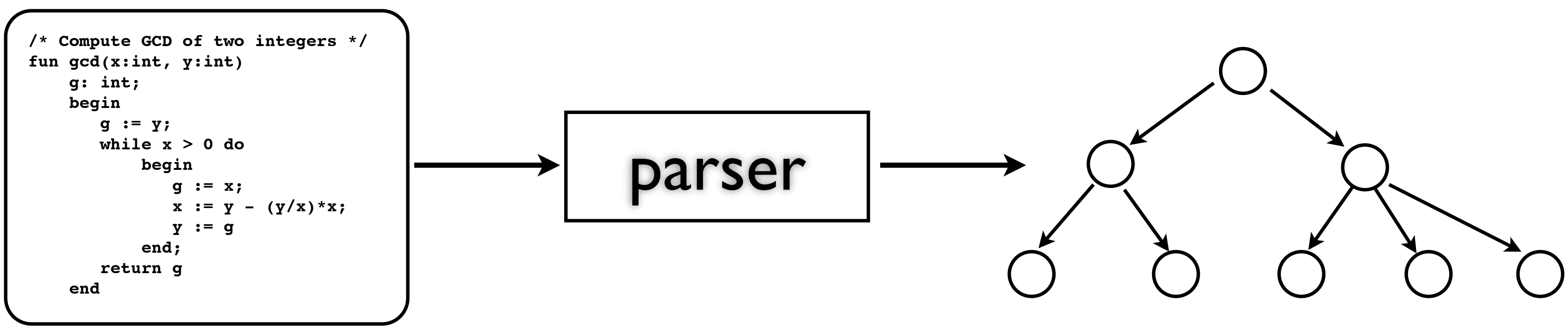
# Example

- Parse and generate assembly code

```
/* Compute GCD of two integers */
fun gcd(x:int, y:int)
    g: int;
    begin
       g := y;
       while x > 0 do
           begin
              g := x;
              x := y - (y/x)*x;
              y := g
           end;
       return g
    end
```

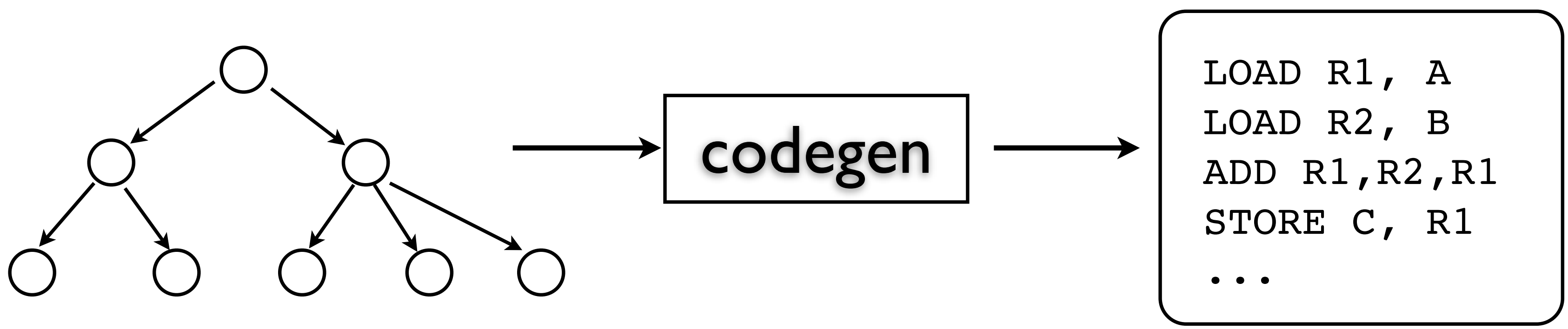
# Compilers 101

- Compilers have multiple phases
- First phase usually concerns "parsing"
- Read program and create abstract representation



# Compilers 101

- Code generation phase
- Process the abstract representation
- Produce some kind of output



# Commentary

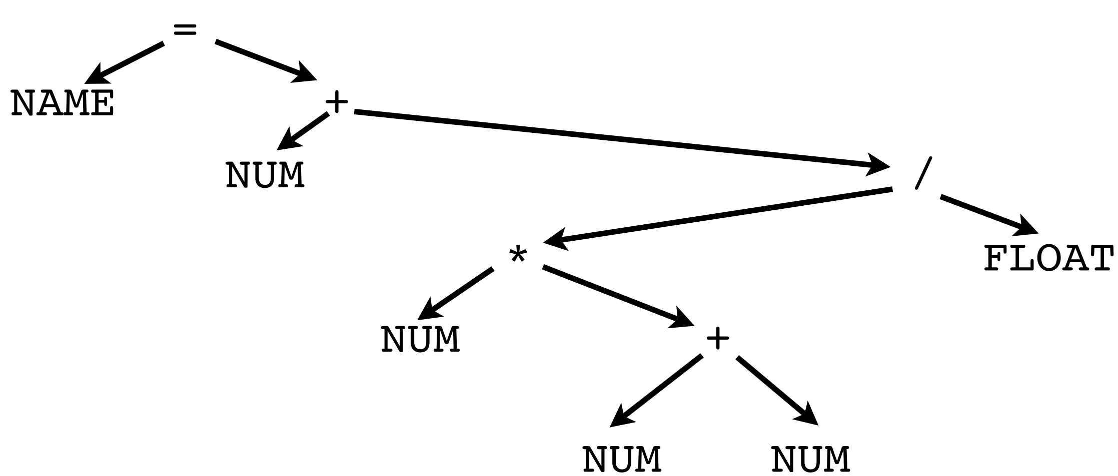
- There are many advanced details
- Most people care about code generation
- Yet, parsing is often the most annoying problem
- A major focus of tool building

# Parsing in a Nutshell

- Lexing : Input is split into tokens

```
b = 40 + 20*(2+3)/37.5

NAME = NUM + NUM * ( NUM + NUM ) / FLOAT
```

- Parsing : Applying language grammar rules



# Lex & Yacc

- Programming tools for writing parsers
- Lex - Lexical analysis (tokenizing)
- Yacc - Yet Another Compiler Compiler (parsing)
- History:

    - Yacc : ~1973. Stephen Johnson (AT&T)
    - Lex : ~1974. Eric Schmidt and Mike Lesk (AT&T)

- Variations of both tools are widely known
- Covered in compilers classes and textbooks

# Lex/Yacc Big Picture

`lexer.l`

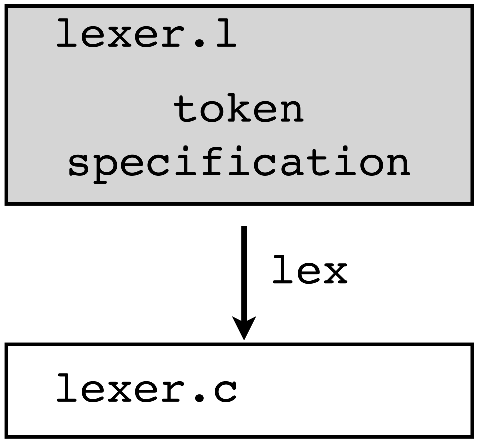
token specification

# Lex/Yacc Big Picture

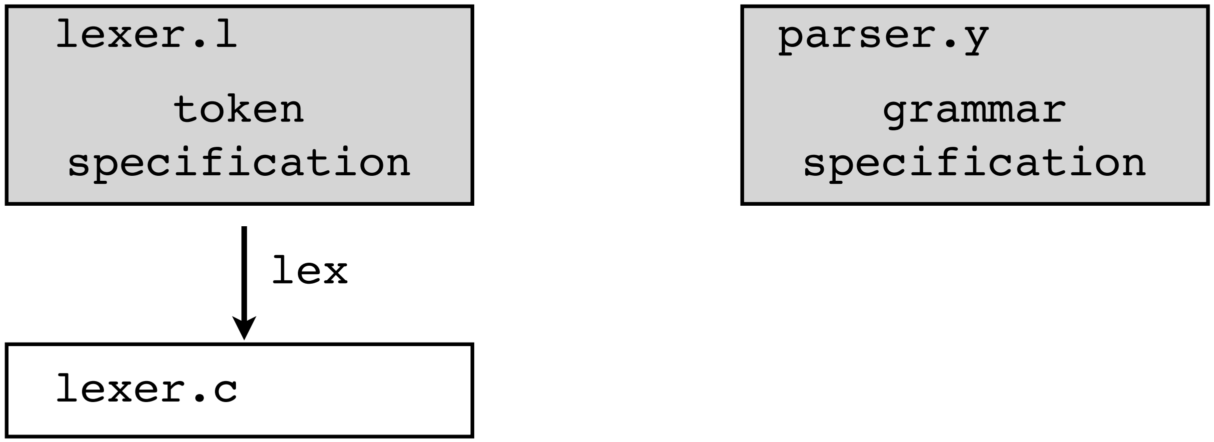
lexer.l

```
/* lexer.l */
%{
#include “header.h”
int lineno = 1;
%}
%%
[ \t]* ;     /* Ignore whitespace */
\n                          { lineno++; }
[0-9]+                      { yylval.val = atoi(yytext);
                              return NUMBER; }
[a-zA-Z_][a-zA-Z0-9_]*      { yylval.name = strdup(yytext);
                              return ID; }
\+                          { return PLUS; }
-                           { return MINUS; }
\*                          { return TIMES; }
\/                          { return DIVIDE; }
=                           { return EQUALS; }
%%
```

# Lex/Yacc Big Picture



# Lex/Yacc Big Picture



# Lex/Yacc Big Picture

lexer.l

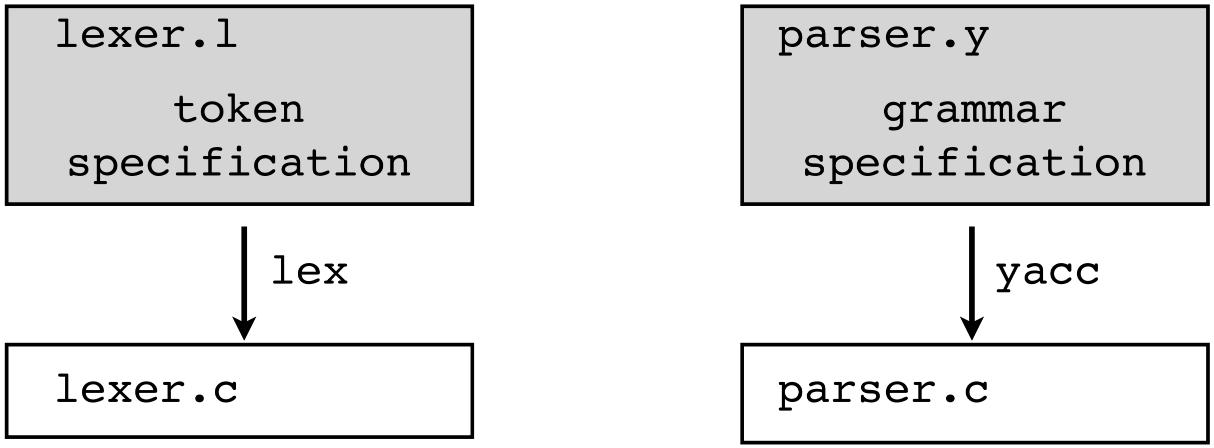
parser.y

spe

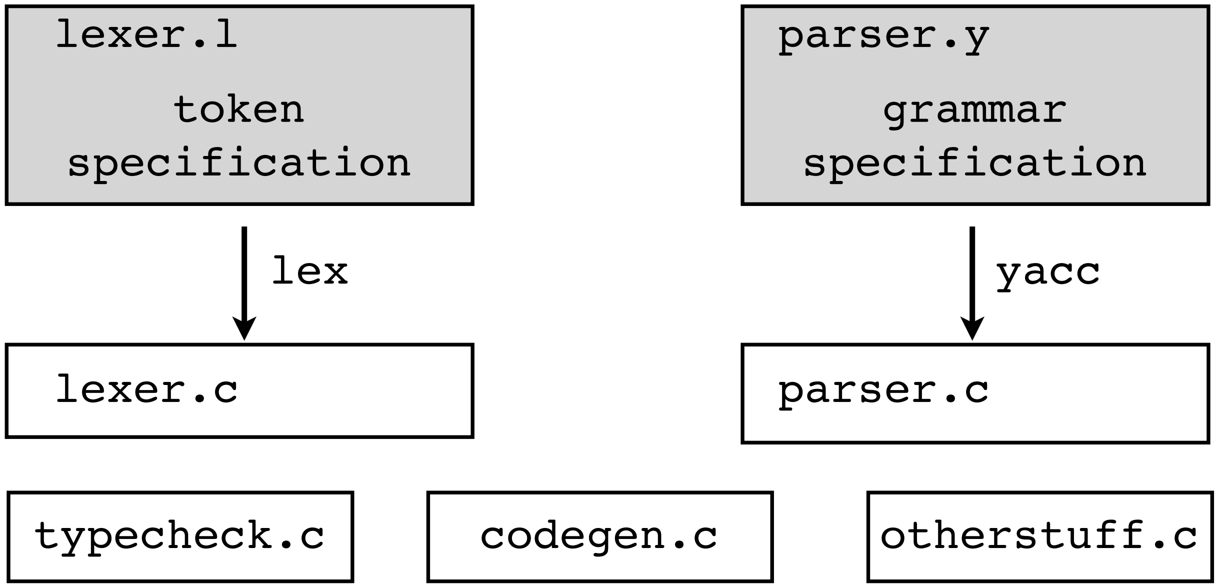
lexe

```
/* parser.y */
%{
#include "header.h"
%}
%union {
   char *name;
   int   val;
}
%token PLUS MINUS TIMES DIVIDE EQUALS
%token<name> ID;
%token<val> NUMBER;
%%
start : ID EQUALS expr;
expr : expr PLUS term
     | expr MINUS term
     | term
     ;
...
```

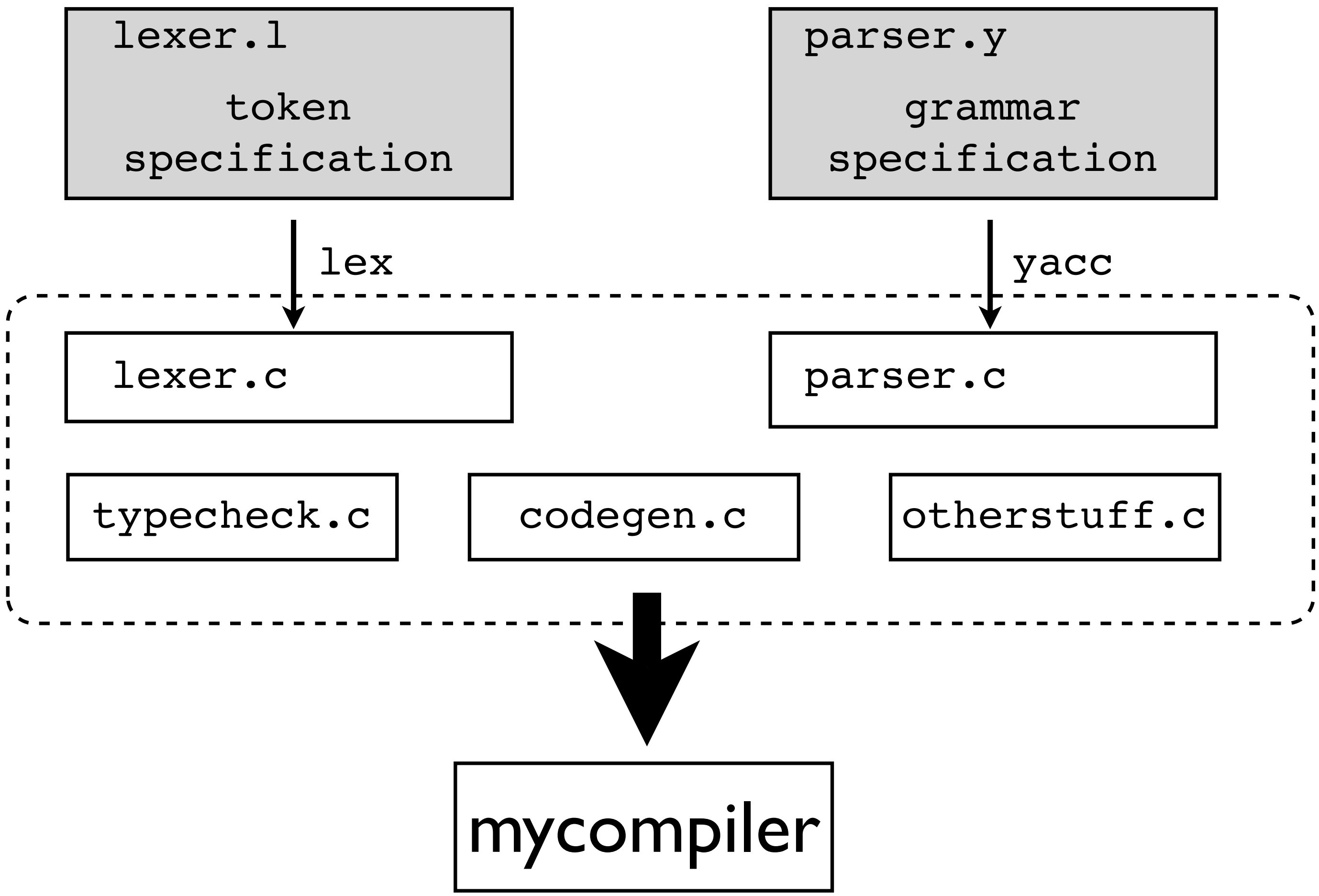
# Lex/Yacc Big Picture



# Lex/Yacc Big Picture



# Lex/Yacc Big Picture



# What is PLY?

- PLY = Python Lex-Yacc
- A Python version of the lex/yacc toolset
- Same functionality as lex/yacc
- But a different interface
- Influences : Unix yacc, SPARK (John Aycock)

# Some History

- Late 90's : "Why isn't SWIG written in Python?"
- 2001 : Taught a compilers course. Students write a compiler in Python as an experiment.
- 2001 : PLY-1.0 developed and released
- 2001-2005: Occasional maintenance
- 2006 : Major update to PLY-2.x.

# PLY Package

- PLY consists of two Python modules

```
ply.lex
ply.yacc
```

- You simply import the modules to use them
- However, PLY is <u>not</u> a code generator

# ply.lex

- A module for writing lexers
- Tokens specified using regular expressions
- Provides functions for reading input text
- An annotated example follows...

# ply.lex example

```
import ply.lex as lex
tokens = [ ‘NAME’,’NUMBER’,’PLUS’,’MINUS’,’TIMES’,
           ’DIVIDE’, EQUALS’ ]
t_ignore = ‘ \t’
t_PLUS   = r’\+’
t_MINUS  = r’-’
t_TIMES  = r’\*’
t_DIVIDE = r’/’
t_EQUALS = r’=’
t_NAME   = r’[a-zA-Z_][a-zA-Z0-9_]*’

def t_NUMBER(t):
    r’\d+’
    t.value = int(t.value)
    return t

lex.lex()         # Build the lexer
```

# ply.lex example

```
import ply.lex as lex
tokens = [ 'NAME','NUMBER','PLUS','MINUS','TIMES',
           'DIVIDE', EQUALS' ]
t_ignore = ' \t'
t_PLUS   = r'\+'
t_MINUS  = r'-'
t_TIMES  = r'\*'
t_DIVIDE = r'/'
t_EQUALS = r'='
t_NAME   = r'[a-zA-Z_][a-zA-Z0-9_]*'

def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t

lex.lex()         # Build the lexer
```

tokens list specifies
all of the possible tokens

# ply.lex example

```
import ply.lex as lex
tokens = [ 'NAME','NUMBER','PLUS','MINUS','TIMES',
           'DIVIDE', EQUALS' ]
t_ignore = ' \t'
t_PLUS   = r'\+'
t_MINUS  = r'-'
t_TIMES  = r'\*'
t_DIVIDE = r'/'
t_EQUALS = r'='
t_NAME   = r'[a-zA-Z_][a-zA-Z0-9_]*'

def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t

lex.lex()          # Build the lexer
```

Each token has a matching declaration of the form t_*TOKNAME*

# ply.lex example

```
import ply.lex as lex
tokens = [ 'NAME','NUMBER','PLUS','MINUS','TIMES',
           'DIVIDE', EQUALS' ]
t_ignore = ' \t'
t_PLUS   = r'\+'
t_MINUS  = r'-'
t_TIMES  = r'\*'
t_DIVIDE = r'/'
t_EQUALS = r'='
t_NAME   = r'[a-zA-Z_][a-zA-Z0-9_]*'

def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t

lex.lex()          # Build the lexer
```

These names must match

# ply.lex example

```
import ply.lex as lex
tokens = [ 'NAME','NUMBER','PLUS','MINUS','TIMES',
           'DIVIDE', EQUALS' ]
t_ignore = ' \t'
t_PLUS   = r'\+'
t_MINUS  = r'-'
t_TIMES  = r'\*'
t_DIVIDE = r'/'
t_EQUALS = r'='
t_NAME   = r'[a-zA-Z_][a-zA-Z0-9_]*'

def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t

lex.lex()         # Build the lexer
```

Tokens are defined by regular expressions

# ply.lex example

```
import ply.lex as lex
tokens = [ 'NAME','NUMBER','PLUS','MINUS','TIMES',
           'DIVIDE', EQUALS' ]
t_ignore = ' \t'
t_PLUS   = r'\+'
t_MINUS  = r'-'
t_TIMES  = r'\*'
t_DIVIDE = r'/'
t_EQUALS = r'='
t_NAME   = r'[a-zA-Z_][a-zA-Z0-9_]*'

def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t

lex.lex()         # Build the lexer
```

For simple tokens, strings are used.

# ply.lex example

```
import ply.lex as lex
tokens = [ ‘NAME’,’NUMBER’,’PLUS’,’MINUS’,’TIMES’,
           ’DIVIDE’, EQUALS’ ]
t_ignore = ‘ \t’
t_PLUS   = r’\+’
t_MINUS  = r’-’
t_TIMES  = r’\*’
t_DIVIDE = r’/’
t_EQUALS = r’=’
t_NAME   = r’[a-zA-Z_]

def t_NUMBER(t):
    r’\d+’
    t.value = int(t.value)
    return t

lex.lex()          # Build the lexer
```

Functions are used when special action code must execute

# ply.lex example

```
import ply.lex as lex
tokens = [ 'NAME','NUMBER','PLUS','MINUS','TIMES',
           'DIVIDE', EQUALS' ]
t_ignore = ' \t'
t_PLUS   = r'\+'
t_MINUS  = r'-'
t_TIMES  = r'\*'
t_DIVIDE = r'/'
t_EQUALS = r'='
t_NAME   = r'[a-zA-Z_][a-zA-Z0-9_]*'

def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t

lex.lex()         # Build the lexer
```

docstring holds
regular expression

# ply.lex example

```
import ply.lex as lex
tokens = [ 'NAME','NUM                                           ,
           'DIVIDE', E
t_ignore = ' \t'
t_PLUS   = r'\+'
t_MINUS  = r'-'
t_TIMES  = r'\*'
t_DIVIDE = r'/'
t_EQUALS = r'='
t_NAME   = r'[a-zA-Z_][a-zA-Z0-9_]*'

def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t

lex.lex()          # Build the lexer
```

Specifies ignored characters between tokens (usually whitespace)

# ply.lex example

```
import ply.lex as lex
tokens = [ 'NAME','NUMBER','PLUS','MINUS','TIMES',
           'DIVIDE', EQUALS' ]
t_ignore = ' \t'
t_PLUS   = r'\+'
t_MINUS  = r'-'
t_TIMES  = r'\*'
t_DIVIDE = r'/'
t_EQUALS = r'='
t_NAME   = r'[a-zA-Z_][a-zA-Z0-9_]*'

def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t

lex.lex()
```

Builds the lexer by creating a master regular expression

# ply.lex example

```
import ply.lex as lex
tokens = [ 'NAME','NUMBER','PLUS','MINUS','TIMES',
           'DIVIDE', EQUALS' ]
t_ignore = ' \t'
t_PLUS   = r'\+'
t_MINUS  = r'-'
t_TIMES  = r'\*'
t_DIVIDE = r'/'
t_EQUALS = r'='
t_NAME   = r'[a-zA-Z_][a-zA-Z0-9_]*'

def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t

lex.lex()         # Build the lexer
```

Introspection used
to examine contents
of calling module.

# ply.lex example

```
import ply.lex as lex
tokens = [ 'NAME','NUMBER','PLUS','MINUS','TIMES',
           'DIVIDE', EQUALS' ]
t_ignore = ' \t'
t_PLUS   = r'\+'
t_MINUS  = r'-'
t_TIMES  = r'\*'
t_DIVIDE = r'/'
t_EQUALS = r'='
t_NAME   = r'[a-zA-Z_][a-zA-Z0-9_]*'

def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value
    return t

lex.lex()          # Build
```

Introspection used
to examine contents
of calling module.

```
__dict__ = {
  'tokens' : [ 'NAME' ...],
  't_ignore' : ' \t',
  't_PLUS' : '\\+',
  ...
  't_NUMBER' : <function ...
}
```

# ply.lex use

- Two functions: input() and token()

```
...
lex.lex()          # Build the lexer
...
lex.input("x = 3 * 4 + 5 * 6")
while True:
     tok = lex.token()
     if not tok: break

     # Use token
     ...
```

# ply.lex use

- Two functions: input() and token()

```
...
lex.lex()          # Build the lexer
...
lex.input("x = 3 * 4 + 5 * 6")
while True:
     tok = lex.token()
     if not tok: break

     # Use token
     ...
```

input() feeds a string into the lexer

# ply.lex use

- Two functions: input() and token()

```
...
lex.lex()          # Build the lexer
...
lex.input("x = 3 * 4 + 5 * 6")
while True:
     tok = lex.token()
     if not tok: break

     # Use token
     ...
```

token() returns the next token or None

# ply.lex use

- Two functions: input() and token()

```
...
lex.lex()          # Build the lexer
...
lex.input("x = 3 * 4 + 5 * 6")
while True:
    tok = lex.token()
    if not tok: break
```

```
tok.type
tok.value
tok.line
tok.lexpos
```

```
          token
```

# ply.lex use

- Two functions: input() and token()

```
...
lex.lex()           # Build the lexer
...
lex.input("x = 3 * 4 + 5 * 6")
while True:
     tok = lex.token()
     if not tok: break
```

```
tok.type
tok.value
tok.line
tok.lexpos
```

token

```
t_NAME   = r'[a-zA-Z_][a-zA-Z0-9_]*'
```

# ply.lex use

- Two functions: input() and token()

```
...
lex.lex()          # Build the lexer
...
lex.input("x = 3 * 4 + 5 * 6")
while True:
     tok = lex.token()
     if not tok: break
```

```
tok.type
tok.value
tok.line
tok.lexpos
```

token

matching text

```
t_NAME   = r'[a-zA-Z_][a-zA-Z0-9_]*'
```

# ply.lex use

- Two functions: input() and token()

```
...
lex.lex()          # Build the lexer
...
lex.input("x = 3 * 4 + 5 * 6")
while True:
     tok = lex.token()
     if not tok: break
```

```
tok.type
tok.value
tok.line
tok.lexpos
```

token

Position in input text

# ply.lex Commentary

- Normally you don't use the tokenizer directly
- Instead, it's used by the parser module

# ply.yacc preliminaries

- ply.yacc is a module for creating a parser
- Assumes you have defined a BNF grammar

```
assign : NAME EQUALS expr
expr   : expr PLUS term
       | expr MINUS term
       | term
term   : term TIMES factor
       | term DIVIDE factor
       | factor
factor : NUMBER
```

# ply.yacc example

```
import ply.yacc as yacc
import mylexer               # Import lexer information
tokens = mylexer.tokens     # Need token list

def p_assign(p):
    '''assign : NAME EQUALS expr'''

def p_expr(p):
    '''expr : expr PLUS term
            | expr MINUS term
            | term'''
def p_term(p):
    '''term : term TIMES factor
            | term DIVIDE factor
            | factor'''
def p_factor(p):
    '''factor : NUMBER'''

yacc.yacc()                # Build the parser
```

# ply.yacc example

```
import ply.yacc as yacc
import mylexer
tokens = mylexer.tokens

def p_assign(p):
    '''assign : NAME EQUALS expr'''

def p_expr(p):
    '''expr : expr PLUS term
            | expr MINUS term
            | term'''
def p_term(p):
    '''term : term TIMES factor
            | term DIVIDE factor
            | factor'''
def p_factor(p):
    '''factor : NUMBER'''

yacc.yacc()              # Build the parser
```

token information imported from lexer

# ply.yacc example

```
import ply.yacc as yacc
import mylexer                  # Import lexer information
tokens = mylexer.tokens         # Need token list

def p_assign(p):
    '''assign : NAME EQUALS expr'''

def p_expr(p):
    '''expr : expr PLUS term
            | expr MINUS term
            | term'''
def p_term(p):
    '''term : term TIMES factor
            | term DIVIDE factor
            | factor'''
def p_factor(p):
    '''factor : NUMBER'''

yacc.yacc()                  # Build the parser
```

grammar rules encoded as functions with names p_*rulename*

Note: Name doesn't matter as long as it starts with p_

# ply.yacc example

```
import ply.yacc as yacc
import mylexer               # Import lexer information
tokens = mylexer.tokens      # Need token list

def p_assign(p):
    '''assign : NAME EQUALS expr'''

def p_expr(p):
    '''expr : expr PLUS term
            | expr MINUS term
            | term'''
def p_term(p):
    '''term : term TIMES factor
            | term DIVIDE factor
            | factor'''
def p_factor(p):
    '''factor : NUMBER'''

yacc.yacc()                 # Build the parser
```

docstrings contain
grammar rules
from BNF

# ply.yacc example

```
import ply.yacc as yacc
import mylexer              # Import lexer information
tokens = mylexer.tokens     # Need token list

def p_assign(p):
    '''assign : NAME EQUALS expr'''

def p_expr(p):
    '''expr : expr PLUS term
            | expr MINUS term
            | term'''
def p_term(p):
    '''term : term TIMES factor
            | term DIVIDE factor
            | factor'''
def p_factor(p):
    '''factor : NUMBER'''

yacc.yacc()
```

Builds the parser using introspection

# ply.yacc parsing

- yacc.parse() function

```
yacc.yacc()      # Build the parser
...
data = "x = 3*4+5*6"
yacc.parse(data)   # Parse some text
```

- This feeds data into lexer
- Parses the text and invokes grammar rules

# A peek inside

- PLY uses LR-parsing. LALR(1)
- AKA: Shift-reduce parsing
- Widely used parsing technique
- Table driven

# General Idea

- Input tokens are shifted onto a parsing stack

```
Stack                          Input
                      <----  X = 3 * 4 + 5
NAME                    <----  = 3 * 4 + 5
NAME =                    <----  3 * 4 + 5
NAME = NUM                         * 4 + 5
```

- This continues until a complete grammar rule appears on the top of the stack

# General Idea

- If rules are found, a "reduction" occurs

```
Stack                         Input
                              X = 3 * 4 + 5
NAME                            = 3 * 4 + 5
NAME =                            3 * 4 + 5
NAME = NUM                          * 4 + 5

         reduce    factor : NUM

NAME = factor
```

- RHS of grammar rule replaced with LHS

# Rule Functions

- During reduction, rule functions are invoked

```
def p_factor(p):
    'factor : NUMBER'
```

- Parameter p contains grammar symbol values

```
def p_factor(p):
    'factor : NUMBER'
       ↑        ↑
      p[0]     p[1]
```

# Using an LR Parser

- Rule functions generally process values on right hand side of grammar rule
- Result is then stored in left hand side
- Results propagate up through the grammar
- Bottom-up parsing

# Example: Calculator

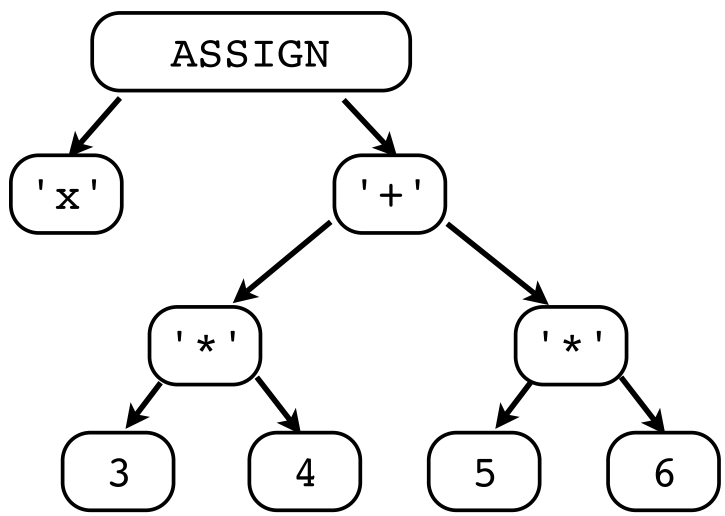
```
def p_assign(p):
    ‘’’assign : NAME EQUALS expr’’’
    vars[p[1]] = p[3]

def p_expr_plus(p):
    ‘’’expr : expr PLUS term’’’
    p[0] = p[1] + p[3]

def p_term_mul(p):
    ‘’’term : term TIMES factor’’’
    p[0] = p[1] * p[3]

def p_term_factor(p):
    '''term : factor'''
    p[0] = p[1]

def p_factor(p):
    ‘’’factor : NUMBER’’’
    p[0] = p[1]
```

# Example: Parse Tree

```
def p_assign(p):
    '''assign : NAME EQUALS expr'''
    p[0] = ('ASSIGN',p[1],p[3])

def p_expr_plus(p):
    '''expr : expr PLUS term'''
    p[0] = ('+',p[1],p[3])

def p_term_mul(p):
    '''term : term TIMES factor'''
    p[0] = ('*',p[1],p[3])

def p_term_factor(p):
    '''term : factor'''
    p[0] = p[1]

def p_factor(p):
    '''factor : NUMBER'''
    p[0] = ('NUM',p[1])
```

# Example: Parse Tree

```
>>> t = yacc.parse("x = 3*4 + 5*6")
>>> t
('ASSIGN','x',('+',
                  ('*',('NUM',3),('NUM',4)),
                  ('*',('NUM',5),('NUM',6))
               )
)
>>>
```



# Why use PLY?

- There are many Python parsing tools
- Some use more powerful parsing algorithms
- Isn't parsing a "solved" problem anyways?

# PLY is Informative

- Compiler writing is hard
- Tools should not make it even harder
- PLY provides extensive diagnostics
- Major emphasis on error reporting
- Provides the same information as yacc

# PLY Diagnostics

- PLY produces the same diagnostics as yacc
- Yacc

```
% yacc grammar.y
4 shift/reduce conflicts
2 reduce/reduce conflicts
```

- PLY

```
% python mycompiler.py
yacc: Generating LALR parsing table...
4 shift/reduce conflicts
2 reduce/reduce conflicts
```

- PLY also produces the same debugging output

# Debugging Output

```
Grammar

Rule 1     statement -> NAME = expression
Rule 2     statement -> expression
Rule 3     expression -> expression + expression
Rule 4     expression -> expression - expression
Rule 5     expression -> expression * expression
Rule 6     expression -> expression / expression
Rule 7     expression -> NUMBER

Terminals, with rules where they appear

*                    : 5
+                    : 3
-                    : 4
/                    : 6
=                    : 1
NAME                 : 1
NUMBER               : 7
error                :

Nonterminals, with rules where they appear

expression           : 1 2 3 3 4 4 5 5 6 6
statement            : 0

Parsing method: LALR

state 0

    (0) S' -> . statement
    (1) statement -> . NAME = expression
    (2) statement -> . expression
    (3) expression -> . expression + expression
    (4) expression -> . expression - expression
    (5) expression -> . expression * expression
    (6) expression -> . expression / expression
    (7) expression -> . NUMBER

    NAME            shift and go to state 1
    NUMBER          shift and go to state 2


    expression                     shift and go to state 4
    statement                      shift and go to state 3

state 1

    (1) statement -> NAME . = expression

    =               shift and go to state 5
```

```
state 10

    (1) statement -> NAME = expression .
    (3) expression -> expression . + expression
    (4) expression -> expression . - expression
    (5) expression -> expression . * expression
    (6) expression -> expression . / expression

    $end            reduce using rule 1 (statement -> NAME = expression .)
    +               shift and go to state 7
    -               shift and go to state 6
    *               shift and go to state 8
    /               shift and go to state 9


state 11

    (4) expression -> expression - expression .
    (3) expression -> expression . + expression
    (4) expression -> expression . - expression
    (5) expression -> expression . * expression
    (6) expression -> expression . / expression

  ! shift/reduce conflict for + resolved as shift.
  ! shift/reduce conflict for - resolved as shift.
  ! shift/reduce conflict for * resolved as shift.
  ! shift/reduce conflict for / resolved as shift.
    $end            reduce using rule 4 (expression -> expression - expression .)
    +               shift and go to state 7
    -               shift and go to state 6
    *               shift and go to state 8
    /               shift and go to state 9

  ! +               [ reduce using rule 4 (expression -> expression - expression .) ]
  ! -               [ reduce using rule 4 (expression -> expression - expression .) ]
  ! *               [ reduce using rule 4 (expression -> expression - expression .) ]
  ! /               [ reduce using rule 4 (expression -> expression - expression .) ]
```

# Debugging Output

```
...
state 11

    (4) expression -> expression - expression .
    (3) expression -> expression . + expression
    (4) expression -> expression . - expression
    (5) expression -> expression . * expression
    (6) expression -> expression . / expression

  ! shift/reduce conflict for + resolved as shift.
  ! shift/reduce conflict for - resolved as shift.
  ! shift/reduce conflict for * resolved as shift.
  ! shift/reduce conflict for / resolved as shift.
    $end            reduce using rule 4 (expression -> expression - expression .)
    +               shift and go to state 7
    -               shift and go to state 6
    *               shift and go to state 8
    /               shift and go to state 9

  ! +               [ reduce using rule 4 (expression -> expression - expression .) ]
  ! -               [ reduce using rule 4 (expression -> expression - expression .) ]
  ! *               [ reduce using rule 4 (expression -> expression - expression .) ]
  ! /               [ reduce using rule 4 (expression -> expression - expression .) ]
...
```

```
    =               shift and go to state 5
```

# PLY Validation

- PLY validates all token/grammar specs
- Duplicate rules
- Malformed regexs and grammars
- Missing rules and tokens
- Unused tokens and rules
- Improper function declarations
- Infinite recursion

# Error Example

```
import ply.lex as lex
tokens = [ ‘NAME’,’NUMBER’,’PLUS’,’MINUS’,’TIMES’,
           ’DIVIDE’, EQUALS’ ]
t_ignore = ‘ \t’
t_PLUS   = r’\+’
t_MINUS  = r’-’
t_TIMES  = r’\*’
t_DIVIDE = r’/’
t_EQUALS = r’=’
t_NAME   = r’[a-zA-
t_MINUS  = r'-'
t_POWER  = r'\^'

def t_NUMBER():
    r’\d+’
    t.value = int(t.value)
    return t

lex.lex()         # Build the lexer
```

```
example.py:12: Rule t_MINUS redefined.
               Previously defined on line 6
```

# Error Example

```
import ply.lex as lex
tokens = [ ‘NAME’,’NUMBER’,’PLUS’,’MINUS’,’TIMES’,
           ’DIVIDE’, EQUALS’ ]
t_ignore = ‘ \t’
t_PLUS   = r’\+’
t_MINUS  = r’-’
t_TIMES  = r’\*’
t_DIVIDE = r’/’
t_EQUALS = r’=’
t_NAME   = r’[a-zA-Z_][a-zA-Z0-9_]*’
t_MINUS  = r'-'
t_POWER  = r'\^'

def t_NUMBER():
    r’\d+’
    t.value = int(t.value)
    return t

lex.lex()         # Build the lexer
```

```
lex: Rule 't_POWER' defined for an
unspecified token POWER
```

# Error Example

```
import ply.lex as lex
tokens = [ 'NAME','NUMBER','PLUS','MINUS','TIMES',
           'DIVIDE', EQUALS' ]
t_ignore = ' \t'
t_PLUS   = r'\+'
t_MINUS  = r'-'
t_TIMES  = r'\*'
t_DIVIDE = r'/'
t_EQUALS = r'='
t_NAME   = r'[a-zA-Z_][a-zA-Z0-9_]*'
t_MINUS  = r'-'
t_POWER  = r'\^'

def t_NUMBER():
    r'\d+'
    t.value = int(t.value)
    return t

lex.lex()         # Build the lexer
```

```
example.py:15: Rule 't_NUMBER' requires
an argument.
```

# Commentary

- PLY was developed for classroom use
- Major emphasis on identifying and reporting potential problems
- Report errors rather that fail with exception

# PLY <u>is</u> Yacc

- PLY supports all of the major features of Unix lex/yacc
- Syntax error handling and synchronization
- Precedence specifiers
- Character literals
- Start conditions
- Inherited attributes

# Precedence Specifiers

- Yacc

```
%left PLUS MINUS
%left TIMES DIVIDE
%nonassoc UMINUS
...
expr : MINUS expr %prec UMINUS {
    $$ = -$1;
}
```

- PLY

```
precedence = (
   ('left','PLUS','MINUS'),
   ('left','TIMES','DIVIDE'),
   ('nonassoc','UMINUS'),
)
def p_expr_uminus(p):
   'expr : MINUS expr %prec UMINUS'
   p[0] = -p[1]
```

# Character Literals

- Yacc

```
expr : expr '+' expr { $$ = $1 + $3; }
     | expr '-' expr { $$ = $1 - $3; }
     | expr '*' expr { $$ = $1 * $3; }
     | expr '/' expr { $$ = $1 / $3; }
     ;
```

- PLY

```
def p_expr(p):
    '''expr : expr '+' expr
            | expr '-' expr
            | expr '*' expr
            | expr '/' expr'''
    ...
```

# Error Productions

- Yacc

```
funcall_err : ID LPAREN error RPAREN {
                  printf("Syntax error in arguments\n");
          }
          ;
```

- PLY

```
def p_funcall_err(p):
    '''ID LPAREN error RPAREN'''
    print "Syntax error in arguments\n"
```

# Commentary

- Books and documentation on yacc/bison used to guide the development of PLY
- Tried to copy all of the major features
- Usage as similar to lex/yacc as reasonable

# PLY is Simple

- Two pure-Python modules. That's it.
- Not part of a "parser framework"
- Use doesn't involve exotic design patterns
- Doesn't rely upon C extension modules
- Doesn't rely on third party tools

# PLY is Fast

- For a parser written entirely in Python
- Underlying parser is table driven
- Parsing tables are saved and only regenerated if the grammar changes
- Considerable work went into optimization from the start (developed on 200Mhz PC)

# PLY Performance

- Example: Generating the LALR tables
  - Input: SWIG C++ grammar
  - 459 grammar rules, 892 parser states
  - 3.6 seconds (PLY-2.3, 2.66Ghz Intel Xeon)
  - 0.026 seconds (bison/ANSI C)
- Fast enough not to be annoying
- Tables only generated once and reused

# PLY Performance

- Parse file with 1000 random expressions (805KB) and build an abstract syntax tree
  - PLY-2.3 : 2.95 sec, 10.2 MB (Python)
  - YAPPS2 : 6.57 sec, 32.5 MB (Python)
  - PyParsing : 13.11 sec, 15.6 MB (Python)
  - ANTLR : 53.16 sec, 94 MB (Python)
  - SPARK : 235.88 sec, 347 MB (Python)
- System: MacPro 2.66Ghz Xeon, Python-2.5

# PLY Performance

- Parse file with 1000 random expressions (805KB) and build an abstract syntax tree
  - PLY-2.3 : 2.95 sec, 10.2 MB (Python)
  - DParser : 0.71 sec, 72 MB (Python/C)
  - BisonGen : 0.25 sec, 13 MB (Python/C)
  - Bison : 0.063 sec, 7.9 MB (C)
- 12x slower than BisonGen (mostly C)
- 47x slower than pure C
- System: MacPro 2.66Ghz Xeon, Python-2.5

# Perf. Breakdown

- Parse file with 1000 random expressions (805KB) and build an abstract syntax tree
  - Total time : 2.95 sec
  - Startup : 0.02 sec
  - Lexing : 1.20 sec
  - Parsing : 1.12 sec
  - AST : 0.61 sec
- System: MacPro 2.66Ghz Xeon, Python-2.5

# Advanced PLY

- PLY has many advanced features
- Lexers/parsers can be defined as classes
- Support for multiple lexers and parsers
- Support for optimized mode (python -O)

# Class Example

```
import ply.yacc as yacc

class MyParser:
    def p_assign(self,p):
        '''assign : NAME EQUALS expr'''
    def p_expr(self,p):
        '''expr : expr PLUS term
                | expr MINUS term
                | term'''
    def p_term(self,p):
        '''term : term TIMES factor
                | term DIVIDE factor
                | factor'''
    def p_factor(self,p):
        '''factor : NUMBER'''
    def build(self):
        self.parser = yacc.yacc(object=self)
```

# Experience with PLY

- In 2001, I taught a compilers course
- Students wrote a full compiler
- Lexing, parsing, type checking, code generation
- Procedures, nested scopes, and type inference
- Produced working SPARC assembly code

# Classroom Results

- You <u>can</u> write a real compiler in Python
- Students were successful with projects
- However, many projects were quite "hacky"
- Still unsure about dynamic nature of Python
- May be too easy to create a "bad" compiler

# General PLY Experience

- May be very useful for prototyping
- PLY's strength is in its diagnostics
- Significantly faster than most Python parsers
- Not sure I'd rewrite gcc in Python just yet
- I'm still thinking about SWIG.

# Limitations

- LALR(1) parsing
- Not easy to work with very complex grammars (e.g., C++ parsing)
- Retains all of yacc's black magic
- Not as powerful as more general parsing algorithms (ANTLR, SPARK, etc.)
- Tradeoff : Speed vs. Generality

# PLY Usage

- Current version : Ply-2.3
- >100 downloads/week
- People are obviously using it
- Largest project I know of : Ada parser
- Many other small projects

# Future Directions

- PLY was written for Python-2.0
- Not yet updated to use modern Python features such as iterators and generators
- May update, but not at the expense of performance
- Working on some add-ons to ease transition between yacc <---> PLY.

# Acknowledgements


- Many people have contributed to PLY

  Thad Austin
  Shannon Behrens
  Michael Brown
  Russ Cox
  Johan Dahl
  Andrew Dalke
  Michael Dyck
  Joshua Gerth
  Elias Ioup
  Oldrich Jedlicka
  Sverre Jørgensen
  Lee June
  Andreas Jung
  Cem Karan
  Adam Kerrison
  Daniel Larraz
  David McNab
  Patrick Mezard
  Pearu Peterson
  François Pinard
  Eric Raymond
  Adam Ring
  Rich Salz
  Markus Schoepflin
  Christoper Stawarz
  Miki Tebeka
  Andrew Waters

- Apologies to anyone I forgot

# Resources

- PLY homepage

  `http://www.dabeaz.com/ply`

- Mailing list/group

`http://groups.google.com/group/ply-hack`